DOR Driver Function and Design

Document version 2.21
Describes driver release version V02-03-03

December 7, 2004

John Jacobsen
NPX Designs, Inc.
for LBNL and the IceCube Collaboration

NPX DESIGNS, INC.
1420 W. Edgewater Ave.
Chicago, IL 60660
773 769 0522 (o)
775 254 5992 (f)
www.npxdesigns.com
info@npxdesigns.com

This document can also be found at
http://docushare.icecube.wisc.edu/docushare/dsweb/View/Collection-511

DOR Driver Cheat Sheet

(Make sure `/usr/local/bin` is in your path)

Check out the DOR driver with e.g. version tag V02-00-09:	`cvs co -r V02-00-09 dor-driver`
Compile driver and test programs:	`cd dor-driver/driver; make`
Install driver in /usr/local and configure the system to load the driver when DOM Hub boots:	`su` `make install`
Load DOR driver to allow firmware installation:	`insmod dh.o`
Remove / unload driver from Linux kernel	`rmmod dh`
Install current firmware version(s) on all currently installed DOR cards:	`make installfw`
Install non-DSB version of same:	`make installnodsb`
(Re)load currently installed driver and firmware	`/etc/init.d/dhrc restart`
Configure DOM Hub to start driver at boot time	`make chkconfig`
Show current firmware and driver versions:	`fvers.pl`
Power on all DOMs attached to the DOM Hub:	`on all`
Power off all DOMs:	`off all`
Interact with DOM boot program (configboot or iceboot) on the B DOM at card 3, wire pair 1:	`domterm 31B` (Control-C to exit)
Put the A DOM at card 0, wire pair 0 into Iceboot from Configboot:	`se.pl 00A r r`
Softboot DOM 7 1 B:	`sb.pl 71b`
Put DOM 1 0 A into echo-mode to prepare for echo test:	`se.pl 10a echo-mode echo-mode`
Perform 10,000-message maximum-throughput echo test on DOM 3 0 B:	`readwrite HUB 30b -s 10000`
For other driver/firmware testing options, see Ref. [2]	

Table of Contents

Introduction

The IceCube experiment, currently being commissioned at the South Pole, is a large particle detector designed to detect faint light signatures from extraterrestrial neutrinos. The detector consists of an array of about 5000 Digital Optical Modules (DOMs) deployed deep in the ice. Each DOM consists of a large photomultiplier tube and an embedded processor board which communicates with a distributed data acquisition (DAQ) system on the surface.

In hardware terms, the surface DAQ architecture consists entirely of consumer, off-the-shelf equipment with one exception: the control and communications interface attached to the long cables which lead to the DOMs. This interface consists of a custom DOM Readout (DOR) card and an associated Linux device driver which is the focus of this document.

IceCube will consist of 80 "strings" of 60 sensors each. Each string will be attached to a single "DOM Hub" consisting of a passive PCI backplane, a dual-processor CPU card, and eight DOR cards (8 DOMs per DOR card, with 4 spare channels per DOM Hub). A single driver must therefore be able to control up to 64 DOMs with a high degree of concurrency.

This document provides a detailed specification of the functionality and design of the device driver for the DOR cards. It will be of interest both to programmers using the driver and to people troubleshooting the hardware and software at multiple levels of the IceCube DAQ. The primary application using the driver is a Java program called DOMHubApp which exposes the relevant functions of the string to an external network interface. Other applications written in C or in scripting languages (Python, Perl) are also used extensively to test the driver, DOR cards and DOM hardware and software.

Additional references (listed at the end of this document) may also be helpful. Basic device driver concepts are covered in much more detail in Reference 1. The device driver design and functionality depends very heavily on the DOR firmware, which is described in Reference 3.

The Role of the Device Driver

The DOR driver provides a single, self-contained interface to 8 DOR cards, and, through that hardware, implements a reliable communications and control interface to 64 Digital Optical Modules.

The DOM Hub runs a commercial distribution of the Linux operating system with a kernel version of 2.4. (As of this writing, the OS version is Red Hat 9.0. The driver has yet to be tested against the brand-new 2.6 kernel.)

The driver runs as a "kernel module," meaning that it is an extension of the Linux operating system which can be loaded at run time. DOM Hub application programs use the driver to control and communicate with the DOR cards and, through these cards, the DOMs deployed deep in the ice. The driver must handle communications through and control of all the DOR cards concurrently. Messages received from and sent to the DOMs are buffered in both directions for maximal performance.

At the lowest level, the driver's operations consist of reading and writing of firmware (FPGA) registers on the cards via the PCI bus, as well as responding to interrupts generated by the DOR hardware. The complexities of buffering, register manipulation and interrupt handling are all hidden by the driver, which provides a straightforward interface for the application programmer via the Linux filesystem.

Driver and Firmware Installation

The following sections explain how the device driver is meant to be built, installed and used.

Building the Driver

The device driver and supporting files are stored in the `dor-driver` project in the IceCube software repository on `glacier.lbl.gov`. The principle source files for the DOM Hub device driver are called `dh.c` and `dh.h`. in the subdirectory `driver`. The other files in the directory are used for testing the driver. DOR firmware versions are stored in `dor-driver/resources/dor-fpga`. Documentation is kept both in `dor-driver/resources/dor-driver-doc` and in the Icecube Docushare document repository.

! **IMPORTANT**: When using BFD or CVS to upgrade to a new release of the DOR driver, it is important to do so from the `dor-driver` directory rather than from the `driver` subdirectory, so you will also get the latest firmware upgrades and driver documentation.

The build tools in the IceCube Software Development Evironment (Ref.) are actually not the best fit for the driver, since that environment is optimized for the development of stand-alone, cross-platform software. The driver, on the other hand, requires platform-specific kernel headers and is meant to be built only on the Linux OS on the DOM Hub. Therefore, a simple Makefile is more appropriate.

The driver, when compiled, consists of a single object file `dh.o` which is to be loaded into the kernel. To compile the driver and associated test programs,

```
% make
```

! **IMPORTANT**: You must have the kernel headers on your machine to build the driver correctly. To find out if you have these,

```
% ls /lib/modules/`uname -r`/build/include
```

If you see a "No such file or directory" error, then you'll have to get your system administrator to put the kernel source tree on the machine, or do it yourself.

Installing the Driver

To install the driver files into the appropriate directories, first build the driver as above, and then

```
% su
# make install
# make chkconfig
```

You should see output showing where the various driver files are installed. In addition to the actual driver file `dh.o`, you will see the latest firmware version, named (as a symbolic link) `dor-current.rbf`. There is also an "rc" script `dhrc` which enables the driver to be installed into the kernel when the DOM Hub boots. The "`make chkconfig`" step sets up the appropriate links for this file in `/etc/init.d`.

Driver Startup and Initialization Parameters

The driver is loaded into the Linux kernel using the `insmod` command. If you did "`make chkconfig`" as described above, this will be done automatically when the system reboots. To load the driver by hand, become the superuser. Then use the insmod command:

```
# insmod /usr/local/dor/driver/dh.o
```

This starts the driver in its default mode of operation. Use the `lsmod` command to see if the driver is installed:

```
# lsmod | grep dh
dh                 217248   0
#
```

Installing the Firmware

When installing a new release of the DOR driver, it is important to update the firmware on the DOR cards as well. The firmware is kept in the directory `dor-driver/resources/dor-fpga` and is upgraded when you do an update of the `dor-driver` project.

The firmware upgrade must be done with the driver loaded in the kernel as described above. To upgrade the DOR firmware, first be sure that the driver is loaded as explained above. Then,

```
# make installfw
```

after `make install` and `make chkconfig`, above. Example:

```
% su
# make install
...
# make chkconfig
...
# insmod /usr/local/dor/driver/dh.o       [if driver is not already loaded]
# make installfw
./installfirmware.pl -rev0 /usr/local/dor/fpga/dor_010q.rbf            \
                     -rev1 /usr/local/dor/fpga/com_100e.rbf            -page 1
Card 0 is DOR Rev 1 <- /usr/local/dor/fpga/com_100e.rbf
Card 1 is DOR Rev 1 <- /usr/local/dor/fpga/com_100e.rbf
Card 2 is DOR Rev 0 <- /usr/local/dor/fpga/dor_010q.rbf
Card 3 is DOR Rev 0 <- /usr/local/dor/fpga/dor_010q.rbf
Card 4 is DOR Rev 0 <- /usr/local/dor/fpga/dor_010q.rbf
Card 5 is DOR Rev 0 <- /usr/local/dor/fpga/dor_010q.rbf
Card 6 is DOR Rev 0 <- /usr/local/dor/fpga/dor_010q.rbf
Card 7 is DOR Rev 0 <- /usr/local/dor/fpga/dor_010q.rbf
Proceed with firmware installation? y
/usr/local/dor/fpga/com_100e.rbf -> /proc/driver/domhub/card0/flash1...Done.
Card 0 reload...Done: new firmware is 100e
/usr/local/dor/fpga/com_100e.rbf -> /proc/driver/domhub/card1/flash1...Done.
Card 1 reload...Done: new firmware is 100e
/usr/local/dor/fpga/dor_010q.rbf -> /proc/driver/domhub/card2/flash1...Done.
Card 2 reload...Done: new firmware is 00aq
/usr/local/dor/fpga/dor_010q.rbf -> /proc/driver/domhub/card3/flash1...Done.
Card 3 reload...Done: new firmware is 00aq
/usr/local/dor/fpga/dor_010q.rbf -> /proc/driver/domhub/card4/flash1...Done.
Card 4 reload...Done: new firmware is 00aq
/usr/local/dor/fpga/dor_010q.rbf -> /proc/driver/domhub/card5/flash1...Done.
Card 5 reload...Done: new firmware is 00aq
/usr/local/dor/fpga/dor_010q.rbf -> /proc/driver/domhub/card6/flash1...Done.
Card 6 reload...Done: new firmware is 00aq
/usr/local/dor/fpga/dor_010q.rbf -> /proc/driver/domhub/card7/flash1...Done.
Card 7 reload...Done: new firmware is 00aq
Creating default FPGA configuration file so that page 1
is automatically loaded by the driver when you reboot....

Done: 8 cards updated.
./configure_clock.pl external
Writing /usr/local/dor/conf/default_clock_select... Ok.
# exit
%
```

If your DOMHub is missing a “DOMHub Service Board” (DSB) (as of this writing this is a common configuration), you need to use alternate firmware. Instead of `make installfw`, type

```
# make installnodsb
```

Firmware Installation Details

The firmware installation is complicated by the fact that there are multiple versions of the DOR hardware (Rev. 0 and Rev. 1). It is further complicated by the existence of non-DSB configurations. For DOR Rev. 0, separate firmware is needed for non-DSB configurations; for DOR Rev. 1, the selection of DSB vs. non-DSB is handled not by the choice of firmware, but by informing the DOR driver to use either the internal DOR clock source, or the external source provided by the DSB.

The Make targets "`make installfw`" and "`make installnodsb`" (and the scripts they invoke, installfirmware.pl and configure_clock.pl) handle the details for the user. The details are shown in the following table.

	Non-DSB	***DSB***
DOR Rev. 0	**20-series firmware** (e.g., dor_020p.rbf) parm_clock_source=0 (ignored)	**10-series firmware** (e.g., dor_010q.rbf) parm_clock_source=1 (ignored)
DOR Rev. 1	100 series firmware (e.g., com_100e.rbf) **parm_clock_source=0**	100 series firmware (e.g., com_100e.rbf) **parm_clock_source=1**

The DOR hardware relies on FPGA designs stored in one or more of three "pages" in flash. A write-protected, failsafe design is loaded automatically from page 0 when the hardware is powered on. Pages 1 and 2 are available to store more up-to-date firmware. Page 3 is used to store the PCI core firmware [DOR Rev. 1 only). Running `make installfw` runs the Perl script `installfirmware.pl`, which programs the latest firmware into page 1 on all installed DOR cards, and configures the driver startup script to load that firmware when the DOM Hub boots up.

One can use installfirmware.pl to install alternate firmware designs on page 1 or 2. To do this, supply the firmware file and page number as arguments on the command line. For example,

```
# installfirmware.pl my_firmware.rbf 2
```

! **NOTE**: Installing a non-functional FPGA design using this script will prevent the DOR driver from working correctly. If this occurs, you can prevent the automatic loading of the corrupt design by removing the file `/usr/local/dor/conf/default_fpga_page` and rebooting the DOM Hub. This will cause the fail-safe design to be loaded, and the firmware install script can be run again.

The `parm_clock_source` driver parameter is set by the `configure_clock.pl` script when `make installfw` or `make installnodsb` are run. It creates the configuration file `/usr/local/dor/conf/default_clock_select`, which contains either 0 [internal] or 1 [external]. This configuration file is read by `/etc/init.d/dhrc` when the system boots and the DOR driver loads. The internal/external clock source can also be toggled by the `clksel` proc file, described below.

Troubleshooting the Driver

Since the device driver is installed directly into the Linux kernel, it does not make use of the standard input and output streams that most user programs use. Instead, diagnostic messages are written to the system error log using the kernel's `printk()` function. In addition, any anomalous conditions are recorded in the driver and can be viewed by inspecting the `lasterr` proc file.

To see the kernel log messages as they appear, run the following command as superuser:

```
# tail -f /var/log/messages
```

You will then see messages from the device driver (generally prepended with "dh") as well as other system messages. Alternately, the `dmesg` command can be run from any account to show most of the kernel log messages.

Simply typing

```
% cat /proc/driver/domhub/lasterr
```

will show the last error condition (if, in fact, an error has occurred). Viewing the file resets the error message to "no error."

The driver communications statistics files `/proc/driver/domhub/cardX/pairY/domZ/comstat` (only present when the driver is loaded) also shows channel-by-channel driver statistics which can be helpful for diagnosing problems.

Driver Installation Options

For the advanced user, several insmod-time options are available:

Parameter	*Options*	*Meaning*
`parm_fpga_page`	0, 1, 2	Page to load FPGA design from at startup (driver default: don't load new FPGA design; installation default: use page 1)
`parm_block`	0, 1	Turn off/on blocking (default: off)
`parm_rx_dma`	0, 1	Turn off/on DMA for received packets (default: on)
`parm_tx_dma`	0, 1	Turn off/on DMA for transmitted packets (default: on)
`parm_verbose`	0, 1, 2	Verbose level (in /var/log/messages); 0 = silent, 1 = quiet, 2 = noisy
`parm_auto_reset`	0, 1	Turn off/on automatic communications reset in case a hardware timeout is detected (default: on)
`parm_fw_version_check`	0, 1	Turn off/on checking of firmware version compatibility (use 0 when testing new firmware; default: on)
`parm_exclude`	0-7	Use to exclude DOR cards from the driver. Repeatable. E.g., parm_exclude=01237

These are given as part of the insmod command line; e.g.,

```
% insmod dh.o parm_fpga_page=1
```

To permanently change your DOM Hub's configuration, edit `/etc/init.d/dhrc` and insert the appropriate options.

Outline of Interfaces

For those who wish to understand role and function of the device driver, it is constructive to define the important interfaces which describe the layers of communication and control in the DOM Hub. These are, in order of least abstract to most:

The Driver/Firmware Interface. This is the "bottom level" of the device driver. The driver interacts with the DOM Hub card by reading and writing memory-mapped FPGA registers. The details of this interface are specified in Reference .

The Driver System Interface. This is the "top level" of the device driver. The system calls `open()`, `close()`, `read()`, `write()`, and `select()` are used to transmit and receive data. Furthermore, control functions (for

example, time calibration, power on/off) are implemented through "virtual" files in the `/proc` filesystem. The Driver System Interface is described in detail below.

Java Interfaces. The functions of the driver are to be fully encapsulated by a few Java classes which will be used by the data-taking application. These classes will be described briefly below.

Communicating with DOMs

Communicating with the DOMs through the DOM Hub cards is relatively straightforward. The basic unit of communication is the "message", a string of characters up to MAXMSG bytes in length. MAXMSG (currently 4092) is obtained by reading the proc file `/proc/driver/domhub/maxmsg`. One simply opens the appropriate device file, which looks like a regular file in the Unix filesystem, and then reads messages from and writes messages to that file.

A file name convention is used which identifies the card, channel (wire pair), and DOM label (A or B) for each DOM. For example,

`/dev/dhc3w1dB`	DOM Hub card 3, wire pair 1, B-DOM
`/dev/dhc0w3dA`	DOM Hub card 0, wire pair 3, A-DOM

These files are created when the driver is installed using "`make install`" (see above). The files can only be opened when the driver is loaded into the kernel.

Before reading and writing these files, use `open()` just like for a regular file. Of course, the DOM you're interested in talking to has to be powered on. The section "Additional Control Functions," below, explains how to make sure a pair of DOMs is powered on.

The `read()` and `write()` system calls are used to receive and transmit data to and from the DOMs. It is important to note that, whether reading or writing, entire messages (up to MAXMSG bytes) are transferred. That means that, when reading, the user program's buffer must be at least MAXMSG bytes in length, and the program must request MAXMSG bytes to be read, even if less bytes might be returned by `read()` (in the case of a smaller message length).

Both blocking and non-blocking I/O are supported in the DOM Hub driver. The default is non-blocking I/O; this can be changed using the `parm_block` parameter at insmod time (e.g., `insmod dh.o parm_block=1`). When the device is open for non-blocking I/O, you should first call `select()` to make sure there is data to be read, or `read()` will return immediately with no bytes read. When `select()` returns true, you can read one or more bytes from the device. The same idea applies when writing in non-blocking mode – call `select()` to be sure that the output buffer isn't full and waiting to be emptied.

DOM Control Functions

Control functions relate to specific actions with the hardware on the surface, as opposed to communicating with the DOMs. Examples of such actions are turning DOM power on or off, or querying the status of the FPGA on a DOR card. Control functions for this driver are implemented through the `/proc` interface. The most common method of passing control information to and from Unix/Linux device drivers is through the `ioctl()` system call. However, since the driver is envisioned to be used by a Java application, it is advantageous to use the `/proc` filesystem for control. This is because reading and writing a file in the filesystem is much more straightforward in Java than calling a system function such as `ioctl()`, which would require a native method interface to a C stub routine rather than the conventional Java input and output functions. Also, `/proc` files can be read or written without the presence of the Java control application, namely by simple shell commands or scripts, which can facilitate testing and debugging.

"Write" functions which cause side effects, such as powering on or off a pair of DOMs, are implemented by writing the appropriate `/proc` file. "Read" functions which get data from the device without side effects (for example, retrieving the local oscillator value for a particular DOM Hub card) are implemented by reading the appropriate file. These files are stored in the top level directory,

```
/proc/driver/domhub
```

which contain all of the control files described in detail below. The DOM Hub proc files are only available when the driver is loaded into the kernel. Once the driver is loaded, proc directories for each installed DOR card appear in the top level directory `/proc/driver/domhub`. Each card has subdirectories for its wire pairs, and further subdirectories for each DOM.

Many of the read functions will return text messages indicating whether or not the requested operation has succeeded. However, proc writes don't have the same ability to communicate the success or failure of their operations. For all operations, especially write operations, check the `lasterr` proc file and make sure the operation succeeded.

A list of proc files and their associated functions follows, starting with functions to program the DOR flash and reload the FPGA from flash.

More on Updating the DOR Card Firmware

Since all communications and control in the DOR card ultimately depends on the manipulation of FPGA registers, it is of crucial importance to be able to upgrade the FPGA firmware. The firmware installation procedure described above uses files in the `/proc` interface to accomplish this.

The DOR card contains a 1 MB flash chip (an AMD 29LV800B) divided into four 256 KB "pages." Each of these pages can contain one FPGA image (241 KB). Since access to the flash memory is typically done through the FPGA, it is important to have a write protected, "fail safe" FPGA design which can always be used to reprogram the flash. By convention, page 0 contains this design. Pages 1 and two can contain alternative, updated FPGA designs. The fail safe design in page 0 is loaded when the power is cycled; any of the alternative pages can be loaded at a later time. By convention, page 3 can be used as "scratch space" to verify that a new design can read and write to the flash. (Page 0 will typically be modified directly with a JTAG interface, though it is possible to do it with the driver; contact the author if you want to do this.)

Here is the sequence of steps for programmng the flash (done automatically when you `make installfw`):

- Send the design into flash page *X* (*X* is 1..2)
- Tell the FPGA to reload itself from flash page *X*
- Verify that the new design is running.

The `/proc` interface which implements these steps is illustrated in the following steps, where we load `DC_004d.rbf` (a raw binary FPGA design file) from the current directory into flash page 1 on card 0.

```
# Tell the flash to accept data:
% echo "flash 1 program" > /proc/driver/domhub/card0/flash1

# Write the FPGA design to flash page 1
% cat DC_004d.rbf > /proc/driver/domhub/card0/flash1
```

When one reads back the bytes, one gets back all 256 KB:

```
% cat /proc/driver/domhub/card0/flash1 | wc –c
262144
```

If one wishes, one can compare the first 246784 bytes (241 KB) with what was sent. This is left as an exercise for the reader. We now cause the new flash design to be loaded into the FPGA:

```
# Load flash page 1 into FPGA
% echo "reload 1" > /proc/driver/domhub/card0/fpga
```

```
# See if it worked – show register 31 (FREV)
% cat /proc/driver/domhub/card0/fpga | grep FREV
FREV  0x00000464
```

Here 0464 translates into "4d", which is the correct firmware version. Alternatively,

```
% /usr/local/bin/fvers.pl
Driver $Revision: 1.128 $
/proc/driver/domhub/card0 -- 004d
```

Most of the other control functions provided by the `/proc` interface are a bit more self explanatory. These are as follows:

System-Wide Proc Functions

Files in **`/proc/driver/domhub/:`**

status: Get current driver status

File: `status` (one file)
Read operation: reports the current status (messages sent and received, etc.).
Write operation: none.

revision: Get current driver revision number

File: `revision` (one file)
Read operation: reports the current RCS revision number (NOT release or "`BFD deliver`'ed" version) of the driver code `dh.c`.
Write operation: none.

lasterr: Get last driver error number and message

File: `lasterr` (one file)
Read operation: reports the most recent anomalous condition that occurred in the driver.
Write operation: none.

Example:

(Assumes that an FPGA reload operation was attempted while a DOM communications file was opened. This isn't allowed because the FPGA goes into an undefined state during reload.)

```
% cat /proc/driver/domhub/lasterr
1: FPGA reload attempted while communications in progress
% cat /proc/driver/domhub/lasterr
0: no error occured since last query
```

The second "cat" gave no error because the first one reset the error flag.

pwrall: Power on/off all DOMs on the Hub

File: `pwrall` (one file)
Read operation: none (use the individual, pair-wise `pwr` proc files).
Write operation: "`on`" or "`off`" to power on or off all the wire pairs plugged into the Hub. This is a clean way to power on all channels at once; if only one wire pair is needed, use the pair-wise `pwr` proc files.

Card-wise Functions

Files in **`/proc/driver/domhub/card*/:`**

rev: Show DOR hardware revision number

File: `cardN/rev` (8 files)
Read operation: shows DOR hardware revision (0 or 1)

clksel: Set/Show DOR clock source

File: `cardN/clksel` (8 files)
Read operation: shows "`internal`" or "`external`" as clock source
Write operation: write "`internal`" or "`external`" to select clock source. **DOMs should be powered off.**

pci: Examine PCI configuration space for a DOR card

File: `cardN/pci` (8 files)
Read operation: shows current PCI configuration space.
Write operation: restores the PCI configuration space, which is saved when the driver is loaded.

Example:

```
% cat /proc/driver/domhub/card0/pci
PCI configuration space for card 0:
                    Vendor: 0x1234
                    Device: 0x5678
                   Command: 0x0087
                    Status: 0x0480
                  Revision: 0x00
                     Class: 0x118000
               Cache line: 0x00
            Latency timer: 0xf8
              Header type: 0x00
                     BIST: 0x00
   Base address register 0: 0x0000a001
   Base address register 1: 0x00000000
   Base address register 2: 0x00000000
   Base address register 3: 0x00000000
       CardBus CIS Pointer: 0x00000000
       Subsystem vendor ID: 0x0000
       Subsystem Device ID: 0x0000
Expansion ROM Base Address: 0x00000000
```

```
IRQ Line: 0x0a
 IRQ Pin: 0x01
 Min_Gnt: 0xf8
 Max_Lat: 0xf8
```

Your vendor and device numbers should agree with what is shown above.

fpga: Show Snapshot of all FPGA Registers, or reload FPGA

File: `cardN/fpga` (8 files)

Read operation: shows all the FPGA registers of a given card (except FIFOs).
Write operation: reload FPGA from DOR flash (see Updating the DOR Firmware, above).
Example:

```
% cat /proc/driver/domhub/card0/fpga
FPGA registers:
CTRL  0x80300001
GSTAT 0x000003c4
DSTAT 0x00000013
TTSIC 0x00000f0f
RTSIC 0x0002000f
INTEN 0x00000000
DOMS  0x00000c03
MRAR  0x00000000
MRTC  0x00000000
MWAR  0x00000000
MWTC  0x00000000
CKCT  0x00000000
DATE  0x00000000
CURL  0x00000000
DCUR  0x01f40000
FLASH 0xff000000
DOMC  0x00000000
DCREV 0x00000004
FREV  0x00000368
```

The final register value (FREV) can be used to determine the version of the DOR firmware currently running (368 corresponds to DC_003h.rbf; 0x68 = ASCII('h')).

fpga-regs: Set or Read a single FPGA Register

Note: Assumes 1 FPGA per card. Generalize appropriately if multiple FPGAs per card.
File: *`cardN/fpga-regs` (8 files)*

This function essentially bypasses the driver interface so that one can test and debug the FPGA firmware. To read a register, write "`r N`" to the file, where `N` is the register number to be read. Reading the file will then show the result. Reading the file without first writing the request will show "`no data`".

To write a register, write "`w N X`". Here `N` is the register number and `X` is the value to write. `X` is a 32-bit hexidecimal number (do *not* include "0x").

Example, showing the reading and writing of register 0, which contains 255 as its beginning read value and is configured to store a value which can then be read back:

```
# echo "r 0" > /proc/driver/domhub/card1/fpga-regs
# cat /proc/driver/domhub/card1/fpga-regs
FPGA on card 1: register 0 = 268435456 (0x10000000)
# echo "w 0 2" > /proc/driver/domhub/card1/fpga-regs
# cat /proc/driver/domhub/card1/fpga-regs
no data
# echo "r 0" > /proc/driver/domhub/card1/fpga-regs
# cat /proc/driver/domhub/card1/fpga-regs
FPGA on card 1: register 0 = 2 (0x2)
#
```

The Perl script `fpga.pl` installed in `/usr/local/bin` streamlines the reading and writing of FPGA registers; this is helpful for driver or firmware debugging.

Please note that `fpga-regs` is writeable and readable only by root, because it is possible to do very bad things to a PC by writing certain values to firmware registers. For this reason, the `fpga` script can only be run by root.

syncgps: Obtain data required to match DOR clock counter values with GPS/UTC time.

File: `cardN/syncgps` (8 files)

Read operation: retrieve the GPS time string & DOR timer value latched at the GPS 1 pulse-per-second (stored as BINARY data).
Write operation: none

Requirements: 10 MHz DOR firmware, Domhub Service Board (DSB) and GPS clock installed.

You can read the proc file directly by reading 22 bytes from `syncgps` (see data format in Ref. 4), or use the `readgps` program installed with the DOR driver.

The latched data pairs are buffered in a FIFO deep enough to hold 11 pairs. If you don't read out the data in time, the FIFO fills up and some data is lost.

See also the `tcalib` proc file (below).

Example:

```
% readgps -h
Usage: readgps <card_proc_file>
Options:  -d       Show difference in DOR clock ticks
          -o       One-shot (single readout)
E.g., readgps /proc/driver/domhub/card0/syncgps
%
$ readgps /proc/driver/domhub/card0/syncgps
GPS 327:04:41:53 TQUAL(' ' exclnt.,<1us) DOR 00000384867e7677
GPS 327:04:41:54 TQUAL(' ' exclnt.,<1us) DOR 0000038487afa377
GPS 327:04:41:55 TQUAL(' ' exclnt.,<1us) DOR 0000038488e0d077
GPS 327:04:41:56 TQUAL(' ' exclnt.,<1us) DOR 000003848a11fd77
GPS 327:04:41:57 TQUAL(' ' exclnt.,<1us) DOR 000003848b432a77
GPS 327:04:41:58 TQUAL(' ' exclnt.,<1us) DOR 000003848c745777
GPS 327:04:41:59 TQUAL(' ' exclnt.,<1us) DOR 000003848da58477
GPS 327:04:42:00 TQUAL(' ' exclnt.,<1us) DOR 000003848ed6b177
GPS 327:04:42:01 TQUAL(' ' exclnt.,<1us) DOR 000003849007de77
GPS 327:04:42:02 TQUAL(' ' exclnt.,<1us) DOR 0000038491390b77
GPS 327:04:42:03 TQUAL(' ' exclnt.,<1us) DOR 00000384926a3877
GPS 328:17:10:20 TQUAL(' ' exclnt.,<1us) DOR 000005e7f8a2c577
GPS 328:17:10:21 TQUAL(' ' exclnt.,<1us) DOR 000005e7f9d3f277
% readgps -d /proc/driver/domhub/card0/syncgps
GPS 328:17:10:22 TQUAL(' ' exclnt.,<1us) DOR 000005e7fb051f77
GPS 328:17:10:23 TQUAL(' ' exclnt.,<1us) DOR 000005e7fc364c77 dt=20000000 ticks
GPS 328:17:10:24 TQUAL(' ' exclnt.,<1us) DOR 000005e7fd677977 dt=20000000 ticks
GPS 328:17:10:25 TQUAL(' ' exclnt.,<1us) DOR 000005e7fe98a677 dt=20000000 ticks
GPS 328:17:10:26 TQUAL(' ' exclnt.,<1us) DOR 000005e7ffc9d377 dt=20000000 ticks
GPS 328:17:10:27 TQUAL(' ' exclnt.,<1us) DOR 000005e800fb0077 dt=20000000 ticks
GPS 328:17:10:28 TQUAL(' ' exclnt.,<1us) DOR 000005e8022c2d77 dt=20000000 ticks
GPS 328:17:10:29 TQUAL(' ' exclnt.,<1us) DOR 000005e8035d5a77 dt=20000000 ticks
GPS 328:17:10:30 TQUAL(' ' exclnt.,<1us) DOR 000005e8048e8777 dt=20000000 ticks
GPS 328:17:10:31 TQUAL(' ' exclnt.,<1us) DOR 000005e805bfb477 dt=20000000 ticks
GPS 328:17:10:32 TQUAL(' ' exclnt.,<1us) DOR 000005e806f0e177 dt=20000000 ticks
```

Pairwise Functions

Files in **`/proc/driver/domhub/card*/pair*/:`**

pwr: Power both DOMs on pair on or off (read/write)

! NOTE: With the current version of the DOR hardware, it is preferable to use the `pwrall` proc file to power on all DOMs at once, because powering single wire pairs has a small chance of disrupting communications in other channels on the Hub. This is likely to be fixed in the next hardware revision.

File: *`cardN/pairM/pwr (32 files)`*
Read operation: shows power status "on" or "off"
Write operation: powers DOM on or off ("on", "off", "reset")

Example:

```
% cat /proc/driver/domhub/card2/pair0/pwr
Card 2 Pair 0 power status is on.
% echo "off" > /proc/driver/domhub/card2/pair0/pwr
% cat /proc/driver/domhub/card2/pair0/pwr
Card 2 Pair 0 power status is off.
%
```

The "`on`" and "`off`" scripts installed with the driver automate this process; e.g., "`on 1 0`" powers on the DOMs on wire pair 0, card 1.

is-plugged: Determine whether a cable (DOM pair) is plugged in

File: *`cardN/pairM/is-plugged (32 files)`*
Read operation: shows "yes" or "no"
Write operation: none

Example:

```
% cat /proc/driver/domhub/card2/pair0/is-plugged
Card 2 Pair 0 is not plugged in.
```

current: Find the current draw of a DOM pair

File: *`cardN/pairM/current (32 files)`*
Read operation: shows current value in milliamps
Write operation: none

Example:

```
% cat /proc/driver/domhub/card2/pair0/current
Card 2 pair 0 current is +24.444 mA.
```

DOM-specific functions

Files in **`/proc/driver/domhub/card*/pair*/dom*/:`**

id: Get DOM ID

File: `cardN/pairM/domX/id` (64 files)
Read operation: communicates with DOM to retrieve 12 byte, hexadecimal DOM ID

Write operation: none

Example:

```
% cat /proc/driver/domhub/card0/pair0/domA/id
Card 0 Pair 0 DOM A ID is 000135871AB2
```

softboot: Perform Soft Reset of DOM

File: `cardN/pairM/domX/softboot` (64 files)
Read operation: none
Write operation: "reset" performs softboot

Example (checking lasterr to make sure softboot succeeds):

```
% cat /proc/driver/domhub/lasterr
0: no error occurred since last query
% echo "reset" > /proc/driver/domhub/card1/pair0/domA/softboot
% cat /proc/driver/domhub/lasterr
0: no error occurred since last query
```

is-communicating: Show Communications State of DOM

File: `cardN/pairM/domX/is-communicating` (64 files)
Read operation: shows if DOR-DOM communications firmware are in a state which allows packet transmission.
Write operation: "`reset`" : Reset communications. Should not be needed, as this is done automatically by the driver if the hardware times out.

Example:

```
% cat /proc/driver/domhub/card0/pair0/domA/is-communicating
Card 0 Pair 0 DOM A is communicating
```

comstat: Show Communications Statistics for DOM

File: `cardN/pairM/domX/comstat` (64 files)
Read operation: shows complete statistics for DOM, including number of messages sent and received, number of hardware packets sent and received, number of ACKs, etc.
Write operation: "`reset`" : Resets counters.

Example:

```
% cat /proc/driver/domhub/card1/pair0/domA/comstat
/dev/dhc1w0dA
RX: 252B, MSGS=20 NINQ=0 PKTS=31 ACKS=7
    BADCRC=0 BADHDR=0 BADSEQ=2 NCTRL=0 NCI=6 NIC=98
TX: 156B, MSGS=7 NOUTQ=0 RESENT=0 PKTS=29 ACKS=22
    NACKQ=0 NRETXB=0 RETXB_BYTES=0 NRETXQ=0 NCTRL=0 NCI=11 NIC=4

    NCONNECTS=5 NHDWRTIMEOUTS=0 OPEN=FALSE CONNECTED=true
    RXFIFO=empty TXFIFO=almost empty,empty DOM_RXFIFO=notfull
%
% echo "reset" > /proc/driver/domhub/card1/pair0/domA/comstat
% cat /proc/driver/domhub/card1/pair0/domA/comstat
/dev/dhc1w0dA
RX: 0B, MSGS=0 NINQ=0 PKTS=0 ACKS=0
    BADCRC=0 BADHDR=0 BADSEQ=0 NCTRL=0 NCI=0 NIC=0
TX: 0B, MSGS=0 NOUTQ=0 RESENT=0 PKTS=0 ACKS=0
    NACKQ=0 NRETXB=0 RETXB_BYTES=0 NRETXQ=0 NCTRL=0 NCI=0 NIC=0

    NCONNECTS=0 NHDWRTIMEOUTS=0 OPEN=FALSE CONNECTED=true
    RXFIFO=empty TXFIFO=almost empty,empty DOM_RXFIFO=notfull
```

tcalib: Perform Time Calibration

File: `cardN/pairM/domX/tcalib` (64 files)
Write operation: Write "`single`" to the proc file to get a time calibration. If another channel on the card is being calibrated, write will return –EAGAIN.

Read operation: Reads the binary time calibration record back. You must first initiate the calibration using the write operation, then read exactly the number of bytes in the calibration record (292), or no data will be returned. If the time calibration isn't ready yet, read will return –EAGAIN.

For synchronizing DOR time stamps to UTC/GPS time, see the `syncgps` proc file, above.

The format of the time calibration record is as follows. The format reflects the maximum number of waveform samples (64); to obtain the actual number of samples used, let *P* equal the low-order 16 bits of the header; the number of samples is then (*P*-32)/2. All numbers are in little-endian byte order.

1. 4 bytes header (0x10098)
2. 8 bytes DOR_t0 (time of pulse generation in DOR)
3. 8 bytes DOR_t3 (time of reply pulse trigger in DOR)
4. 64 * 2 bytes DOM waveform measured in DOR
5. 8 bytes DOM_t1 (time of pulse triggering in DOM)
6. 8 bytes DOM_t2 (time of reply generation in DOM)
7. 64 * 2 bytes DOR waveform measured in DOM

Total 292 bytes

Since the shell command "`cat`" won't work on the time calibration proc file (because of the fixed length record), a program tcaltest.c is included in the driver distribution which performs the write operation, then reads and displays the correct number of bytes for inspection by eye. Generally, however, the data will be read by a user application which will decode the record and do something useful with it.

Example time calibration:

```
% tcaltest $proc/card1/pair3/domB/tcalib
cal(0) dor_tx(0x2fd7ffa90a) dor_rx(0x2fd7ffad2d) dom_rx(0x3530c8385) dom_tx
(0x3530c85e7)
dor_wf(518, 517, 517, 518, 519, 519, 518, 518, 518, 516, 516, 515, 516, 516, 518,
518, 520, 521, 522, 524, 533, 551, 577, 610, 646, 687, 726, 767, 806, 846, 883,
917)
```

```
dom_wf(530, 528, 529, 529, 531, 531, 531, 529, 528, 528, 528, 527, 528, 527, 528,
532, 532, 532, 532, 535, 539, 551, 572, 600, 633, 671, 712, 749, 791, 828, 868,
901)
%
```

In addition to these control functions, other files in `/proc` may be defined which show the status of the driver or help in debugging.

Testing the DOR Driver and Firmware

Several test programs are available and installed as part of the `dor-driver` project. These programs have been written by this author and by Arthur Jones at LBNL. The test procedure is somewhat formalized and is covered at length in Rev. [2].

Java and the DOR Driver

The DOM Hub device driver will be used most commonly by one or more DOM Hub DAQ applications. Since all of the communications and control functions implemented by the driver can be carried out by reading and writing files in the filesystem (either in `/dev` or `/proc`), the normal I/O functions in Java can be used without resort to Java Native Interface (JNI) methods. This feature simplifies application development and allows for easier simulation and testing at the Java Layer level.

A full-fledged data acquisition application will need to communicate with multiple DOMs at the same time (synchronous I/O). Such an application can implement synchronous I/O either by multithreading and using blocking I/O, or by looping over open channels in non-blocking mode (the default). Since select behaviour is not (yet) supported in the Java NIO (Ref.), it remains an open question how to mock-up a selectable channel in the driver; most likely, the reading application will simply retry the read operation on channels with no data available, possibly tuning time delays so as to avoid unnecessary burden on the CPU. In anticipation of this situation, the driver implements non-blocking I/O as the default.

There are a few Java software layers which live above the driver. The communications functions of the driver are implemented in the RealDOMDriver class. Control operations (power on/off, etc.) are implemented in RealCardDriver. These communications and control objects are produced by a factory object call RealDHDriverFactory. Java programs can use these classes to interact with the driver without knowing the details of reading and writing the device files or the `/proc` files.

In addition to RealDOMDriver and RealCardDriver, there are simulation classes which implement the same functionality for network-based simulations of the DOM Hub and DOM components of IceCube. For example, SimSocketDOMDriver implements the analog of RealDOMDriver for the case where one is exchanging messages, not with the real hardware (nor with simulation programs talking through the `dh` running in `sim_queue` mode), but over a network socket to a DOM simulation program (`domapp`). The abstract class called DOMDriver can be used by higher-level DAQ and testing programs which function the same way, regardless of whether they are talking to the real hardware, to simulation programs using the device driver, or a pure simulation running in a distributed, networked computing environment.

These helper classes (`RealDOMDriver`, etc.) are part of the `domhub` project in the IceCube software archive.

Driver Design and Implementation Details

The final sections of this document give details on how the DOR device driver is implemented as well as additional options for its use and debugging. The intended audience is those people who wish to contribute to driver development, or use existing code to help with new projects, or just to understand better how the driver works.

General Remarks about Linux device drivers

The text by Rubini and Corbet (Ref. 1) provides a much more thorough introduction to Linux device drivers than can be provided here. However, a few general principles can be outlined.

A device driver can be thought of as a mechanism within the operating system to connect a user's program to some physical hardware. The driver hides the complexity of the specific operations required to cause the hardware to perform its tasks. The writer of the driver has to understand how the operating system works with processes, memory, files, etc.; the hardware interface in detail, down to the level of individual bits and bytes; and what the user's needs and tools are likely to be.

There are a few different classes of Linux device drivers. The DOR card is implemented as a character device. Character devices are meant to be written to and read from as streams of bytes. Block devices such as disks and network drivers behave somewhat differently. Consult Ref. 1 if you're curious to learn more about such things. (There is a subtlety here: actually the driver operates on chunks of bytes (packets or messages) rather than character streams, but the way the driver is used matches the traditional character driver fairly closely.)

The general behaviour of character device drivers under Linux (and Unix in general) is shaped by a core design philosophy of the Unix operating system, namely, to treat as many things as possible, including physical devices, as files in the heirarchical file system. The files can be opened, closed, read, and written using the same functions one uses to manipulate regular files.

The task of the driver writer, then, is largely to construct the appropriate "callback" functions which are called when the user calls `open()`, `close()`, `read()` and `write()`. In our case, the driver functions corresponding to these standard C library calls are `dh_open()`, `dh_close()`, `dh_read()`, and `dh_write()`. These functions arrange for the transfer of data from priveleged "kernel space" (where the driver functions run) to the application's "user space" buffers. Generally speaking, additional control of the driver can be provided by an implementation of `ioctl()` and/or implementations of `/proc` files. As mentioned earlier, `/proc` files are favored for the DOR driver because of Java's platform-independence restrictions. The DOR driver therefore takes the Unix philosophy of "a file for everything" to the limit!

The callback functions are all packaged up into a compiled bit of C code (a "module") which can be either compiled statically into the kernel or loaded dynamically at run time using the `insmod` command.

The driver also consists of initialization code (`dh_init()`) and cleanup code (`dh_cleanup()`). The initialization code, invoked at `insmod` time, registers the `read`/`write`/`open`/`close` and `/proc` callbacks with the kernel, allocates software copies of write-only FPGA registers, and finds the PCI addresses of available DOR cards in the system. Further initialization such as creation of message queues is done when a user program opens a device file (`/dev/dhc*w*d*` or `/dev/simc*w*d*`) with `dh_open()` in order to communicate with a DOM. These message queues are deallocated when the device file is closed.

It is important to note that, although there are up to 8 DOR cards and up to 64 DOMs on a DOM Hub, there is only one copy of the driver code installed in the kernel. All of the devices share the same callback functions, though each has its own message queues and set of FPGA register copies. This makes sense because all the devices are identical (save for their DOR card IDs) and are handled the same way. The situation reflects the fact that it is unlikely that substantially different DOR firmware designs will be running on the same Hub at the same time.

Driver Options

There is a switch to the driver which allows for one to set blocking or non-blocking I/O. (Traditionally this is done with an `fcntl()` call; however, see the discussion about the interface to the Java layer, below.) The `parm_block` option toggles blocking or non-blocking I/O:

```
# insmod dh.o parm_block=1
```

installs the driver in blocking mode. `parm_block=0` (non-blocking I/O) is the default.

A final parameter of note is `parm_fpga_page`. This can be set to the value 1, 2 or 3 to tell the driver to load alternate DOR flash pages, and is used by `dhrc` when the configuration file `default_fpga_page` is present.

Code Organization

Since the driver is basically a set of callbacks with supporting functions, the structure of the code is pretty straightforward. As mentioned earlier, all the code for the driver resides in a file called `dh.c` with a supporting header file `dh.h`, which defines the relevant numerical constants and FPGA register address locations (relative positions within the PCI resource space). Lower level functions are generally located closer to the top of `dh.c`, while the higher level functions that use them are towards the bottom. To avoid namespace pollution in the case that any symbols are exported, each function and global variable begins with the prefix `dh_`, and most functions and global variables are tagged with the `static` keyword.

PCI Interface to the DOR FPGA Firmware

The DOR card firmware uses a commercially available firmware product to implement the 32-bit PCI specification. In addition to the PCI machinery, the FPGA also implements a set of registers used to control the DOR card and communicate with the DOM. The PCI functionality allows the device driver to use a set of standard kernel functions to discover the location of those registers in memory or I/O space.

The function `dh_get_pci_devices()` performs this discovery process by looping on the kernel function `get_pci_device()`, looking for the appropriate vendor (`DH_VENDOR`) and device (`DH_DEVICEID`) values. It then uses additional PCI functions to "enable" (in the sense of the PCI standard) each device and get its interrupt request line (IRQ) number. It also determines whether the FPGA registers are kept in memory-mapped or I/O space and finds out exactly where they live. The IRQ number and the register locations are stored in a `struct dh_pci_dev` for each device. The functions

```
dh_read_regi()
dh_write_regi()
dh_set_write_regi_bits()
dh_clear_write_regi_bits()
dh_test_read_regi_bit()
```

are then able to use the register locations to read or write the appropriate FPGA registers. The higher-level functions call these functions to accomplish the basic I/O and control tasks of the driver. (NOTE: memory-mode accessing of registers it temporarily disabled; I/O space should be used instead, and this is in fact the way it is done by all versions of the DOR firmware).

The loading and re-loading of the FPGA firmware is somewhat tricky but is explained in Ref. 4. One writes an FPGA image (`.rbf` file) into the DOR flash memory, then pulls on the correct register to initiate a re-load of the firmware from flash. Since the FPGA design implements the PCI functionality, reloading the design causes an "upset" in the PCI configuration registers, which have to be saved before reload and then restored afterwards. This is taken care of in `dh_reload_fpga_from_flash()`. Other functions used for firmware programming and loading are:

`dh_fpga_write_proc()`: implements the `/proc` user interface to `dh_reload_fpga_from_flash()`;

`dh_save_pci_config_space()` and `dh_restore_pci_config_space()`: save and restore PCI configuration during reload;

`dh_flash_write_proc()`: `/proc` callback which allows the user to program a new FPGA design into flash

Messages and Message Queues

Despite the fact that, from the kernel's point of view, the driver in question is a "character device," which suggests that bytes, rather than byte-aggregates, are the fundamental unit of data transfer, the DOR driver in fact transfers "messages" from 1 to MAXMSG bytes in length. This design choice allows one to move the DOM messaging protocol all the way down into the firmware, resulting in a significant savings in processor overhead for both the DOM and the DOM Hub. The decision has several implications for both the driver and the application writer. One nice feature, again, is that many details of the protocol are hidden. However, as stated above, a `read()` request to the driver must ask for the maximum message size, MAXMSG bytes. Less bytes may be returned in the case of a smaller message, but at least that many bytes must be allocated to the buffer in the calling application. If a smaller buffer were allocated, the driver would have nowhere to put a larger message arriving from the DOM.

The chunking of data into messages also affects how data is buffered in the driver. Buffering is crucial because it allows the DOR hardware to transfer data into memory when it is available (i.e. at interrupt time), regardless of whether a user application is reading it at that very moment; and, similarly, it allows a user application to do other things after writing data into the driver, which may take time to transmit the completed message to the DOM due to the restricted bandwidth of the twisted quad cable.

Messages are buffered in circular queues of fixed-length arrays. Each device file has an input and an output queue which are allocated in their entirety when the device file is opened. The queues accommodate 100 output and 100 input messages per device file. This may not seem like much but should be more than adequate since the typical mode of operation in the DOM is "call-and-response" (DOM Hub initiates request with one message, then waits for the DOM's reply)[1]. Although fixed-length queues might seem strange, it is anticipated that the largest message lengths will be the most common (waveform transmission). The choice of pre-allocated queue buffers allows one to avoid the overhead and complexity of repeated memory allocation and de-allocation, at the cost of a larger kernel module size for the driver.

Message queue manipulations are provided by the following functions, some of which are have self-explanatory names:

```
dh_init_message_queue()
dh_queue_isempty()
dh_queue_isfull()
dh_num_in_queue()
dh_release_message_queue()
```

`dh_queue_get()` – read a message from a queue
`dh_queue_add()` – add a message to a queue
`dh_move_message()` – move a message from one queue to another (as described for dh_write(), above)

The queue functions make use of `dh_copy_message()`, which moves whole messages around within kernel space, or from kernel space to user space (see Ref. 1 for a detailed explanation of the difference between user space and kernel space).

[1] In the case where the DOM is running in one of its boot modes (Iceboot or Configboot), many messages in a single direction are possible. A flow control mechanism in the driver, DOR firmware and DOM Boot code ensures that no packets are lost if the user application is too slow to read messages out of the driver's buffers and the buffers become completely filled.

New Communications Protocol

In early 2004, a new protocol was implemented in both the DOR driver and the DOM software which included the detection and correction of communications errors. The scheme was based loosely on certain features of TCP/IP and was designed primarily by Arthur Jones, who implemented the code on the DOM side while Jacobsen implemented it in the DOR driver.

Whereas before there was a one-to-one correspondence between messages written to / read from the DOR driver, and packets sent on the wire by the DOR firmware, the new protocol is somewhat more complicated. Messages larger than 488 bytes is broken up into multiple "hardware packets" and reassembled on the other end. A CRC is calculated for each hardware packet which allows one to detect the occasional bit error introduced by noise in the hardware. Furthermore, upon transmission, each packet is assigned a sequence number. The sequence number and packet type are encoded in an 8 byte packet header, and the 32-bit CRC is appended at the end of the packet. Each hardware packet is acknowleged by the recipient, unless it has a bad CRC or is out of sequence. Packets not receiving ACKs are retransmitted after a small timeout interval. In order to synchronize packet sequence numbers, a synchronization protocol is used whenever a new connection is established (DOM device file opened on the DOM Hub side, or DOM change of state, e.g. from Iceboot to Domapp).

While packet assembly and disassembly are carried out in userland (read and write functions), the transmission, verification, acknowledgement and retransmission are done in the interrupt handler code. (This choice is somewhat unconventional, because in general it is desireable to keep the interrupt handler as simple as possible. However, this approach allowed the maximal reuse of existing code, and in retrospect provided a very clean solution.) A kernel timer handles the resetting of communications after a hardware timeout, and does the resynchronization protocol.

Interrupt Handling

Almost every device driver makes use of interrupts, which allow the hardware to signal the driver that an event has occurred. The most typical example is when the device has received data that should be read out and buffered by the driver. The DOR firmware can provide a variety of interrupts, but the most important are interrupts that signal the presence of data in the RX (receive) FIFO, or the availability of enough space in the TX (transmit) FIFO to receive a message to be sent to the DOM.

A Linux driver handles interrupts by registering an "interrupt handler" with the kernel. The interrupt handler is a subroutine which is called immediately when an interrupt occurs. This handler must determine if the interrupt in fact belongs to it (interrupts can be shared between multiple handlers), and then handle whatever condition occurred to cause the interrupt. This has to happen quickly because the processor will generally not do anything else until the handler exits. Operations that can't be accomplished immediately are typically scheduled to be handled at a later time by a so-called "bottom-half" routine which will be invoked at a time that is more convenient for the kernel, as explained in Ref. 1. In the case of the DOR driver, unfortunately, bottom halves are not scheduled quickly enough to fill or drain the communications FIFOs without overflowing (at least given the current FIFO depths in the DOR firmware), so this is currently done immediately in "bh-like" subroutines called by the interrupt handler.

In the DOR driver, the same handler `dh_intr_hdlr()` gets registered for each DOR card. Since interrupts are shared in the PCI specification, the handler might get registered multiple times for a single IRQ. When a signal occurs on that interrupt line, each handler runs and looks at the firmware registers to see if the interrupt belongs to its card. If it does, it then sees what sort of interrupt it is (using the firmware registers), temporarily disables interrupts of that kind on that card, and executes the appropriate "bh-like" drain or fill routine. If an unknown interrupt occurs, then all interrupts on that card are disabled as a last resort.

The "bh-like" functions for data transmission called by the interrupt handler are `dh_do_send_message()` (for TX) and `dh_do_recv_message()` (for RX). The functions check to see which FIFOs have room to receive data (TX), or have data to read out (RX); perform the appropriate fill/drain operations; and then reenable the appropriate type of interrupt for that card.

All this somewhat low-level stuff connects with the actual reading or writing user program in the following way. When the user program calls `write()`, the message is queued by `dh_write()` and TX interrupts are enabled (they are disabled by default). RX interrupts are enabled by default; when data arrives in the DOR card from the DOM, the interrupt handler uses `dh_do_recv_message()` to queue the message and then, as a last step, awakens any reading process which may have blocked in `read()` / `dh_read()`.

Most drivers which handle interrupts are susceptible to race conditions. These are avoided in the DOM Hub driver by keeping separate data structures for each device file, and by using spinlocks to guarantee mutually exclusive access to shared resources. More on spinlocks and race conditions can be found in Ref. 1.

Additional types of interrupts, such as interrupting when a DOM pair cable is plugged into or unplugged from the DOM Hub, are described in the DOR firmware interface, but they are not (yet) implemented in the driver.

Direct Memory Access (DMA)

DMA is a way of copying blocks of data to and from a peripheral such as the DOR card without having to issue repeated single instructions on the PCI bus. The driver provides the hardware with a direct pointer to a memory location and a count, and the hardware reads from or writes to this memory space without the driver having to do multiple reads or writes on a FIFO. As a result, using DMA can result in significantly better performance. In the case of the DOR driver, DMA is advantageous only for reading out RX messages because they are significantly longer than TX messages (MAXMSG bytes vs. roughly 8-16 bytes).

In August of 2003 I implemented DMA in the receive portion of the DOR driver interrupt handler, and used fine-grained benchmarking tools provided by Andrew Morton (ref. 10) along with a rebuilt Linux kernel to calculate that the DMA version was about three times faster than the non-DMA version for receive. Preliminary numbers from these tests indicate that the DOR driver will provide adequate throughput even for the fully-populated DOM Hub with 60 DOMs each running at a data rate of 2 Mbps or less. DMA for TX was added in October of 2003.

Enabling and disabling DMA can be done using the `parm_tx_dma` and `parm_rx_dma` parameters at ismod time; e.g.,

```
# insmod dh.o parm_tx_dma=1 parm_rx_dma=0
```

will load the driver configured to use DMA for TX but not for RX. Currently, the default is to use DMA for both TX and RX.

Time Calibration

A crucial aspect of the IceCube experiment is the ability to calibrate the free-running DOM clocks against a global time standard with an accuracy of ~5 ns. The problem is solved by sending short time calibration pulses in both directions on the cable and digitizing each at the opposite end, and then collecting the local and remote time-stamps and digitized waveforms of each pulse at the surface. The resulting data can be analyzed to give a very precise measurement of the offset between local (DOM) and global time. The bulk of this work is done in firmware in the DOM and in the DOR card; the driver merely formats the information from the firmware and makes it available to a user application. Currently the time calibration must be initiated “by hand” by writing to the `tcalib` proc file. Eventually, time calibration will occur synchronously during a brief shutdown of communications, and this will be controlled completely by the hardware.

Some care is taken in the driver to guarantee that the time calibration and communications don't interfere with each other: a global spinlock protects the firmware on each card so that time calibration initiation and readout can occur completely asynchronously along with communications, without races.

Implementation of `/proc` Files

As mentioned before, it is most common to use `ioctl()` to perform device control tasks not associated with I/O, but the `/proc` interface allows one to do these tasks more easily with Java programs. Because of the somewhat complicated heirarchical structure of the DOM Hub functions (8 cards supporting 4 wire pairs each; each wire pair supporting 2 DOMs), some extra code was written to help package the `/proc` interface in such a way that adding or removing additional control files or subdirectories would be more straightforward.

The `/proc` filesystem is discussed in detail in Refs. , and . Each file or subdirectory in `/proc` is associated with a `struct proc_dir_entry`. In the DOR driver, this structure is wrapped up along with a few other variables in a `struct fancy_pde_entry`. These extra variables contain the file or directory's name and a flag indicating whether the file was successfully "registered" with the kernel. This allows the function `dh_cleanup_proc_entries()` to be called at any point if the module is removed from the kernel or if initialization fails for some reason; only proc files which have been successfully registered are then unregistered. `dh_initialize_proc_files()` is called at module init time to register all the proc files; it uses the helper functions `dh_make_fpde_as_file()` and `dh_make_fpde_as_dir()` to cleanly register the files/directories. As stated earlier, only cards physically present in the DOM Hub (or detectable using the PCI interface) show up in the proc filesystem. The exception to this rule is, of course, if the device driver is running in simulation mode, in which case proc files are not only shown for cards 0..7, but also 8..15 (the files `/dev/simc`*X*`w`*Y*`d`*Z* correspond to the higher-numbered cards).

Each proc file has optional read and write callback functions which are supplied as arguments to `dh_make_fpde_as_file()`. The function also allocates a data structure of arbitrary size (specified by the `data_size` argument) to be used to pass data to the callback functions. This allows the same callback to be used for multiple proc files. For example, each card has an `fpga` proc file – each of these files uses the callback `dh_fpga_read_proc()` to return status information on the card's FPGA firmware; the callback looks at the data structure passed in through the data argument (`data_size` bytes) to determine which card to report data for.

The callbacks report data (`dh_*_read_proc()`) or get requests (`dh_*_write_proc()`) by transferring data to and from the user program through the `buffer` argument to the callback. `snprintf()` is generally used to report data in the read callbacks and `sscanf()` or `strncmp()` are used to parse the arguments to write requests. Ultimately these result in FPGA registers being read or written to accomplish the requested task, unless the driver is running in simulation mode. In that case, most write functions do nothing, and the read functions attempt to report something reasonable.

References

1. Rubini & J. Corbet, *Linux Device Drivers*, 2nd Ed., 2001. O'Reilly & Associates..

2. J. Jacobsen, DOR Driver and Firmware Testing Procedure, available at `http://docushare.icecube.wisc.edu/docushare/dsweb/View/Collection-511`

3. K. Sulanke, *The DOMCOM FPGA/PLD 8-bit register map*, 2001. `http://www-zeuthen.desy.de/~sulanke/Projects/DOMCOM/Doc/Domcom_API.doc`.

4. K. Sulanke, *DOR-API Description*, 2003. Rev. 25 or later. Description of the FPGA registers used to control the DOR hardware. `http://www-zeuthen.desy.de/~sulanke/Projects/ICECUBE/DOR/doc/`

5. R. Hitchens, *Java NIO*, 1st Ed., 2002. O'Reilly & Associates.

6. S. Patton, *An Introduction to the IceCube Software Development Environment*, 2002. IceCube internal memo.

7. D. van Heesch, *Doxygen project*. `http://www.stack.nl/~dimitri/doxygen/`

8. E. Mouw, *Linux Kernel Procfs Guide*, Rev. 1.1, 2001. Contact `J.A.K.Mouw@its.tudelft.nl` for details.

9. Nicholas McGuire, *The Linux Proc File System for Embedded Systems – Concepts and Programming*, `ftp://ftp.opentech.at/pub/rtlinux/contrib/hofrat/embedded_proc.pdf`, 2003.

10. Andrew Morton's `Timepeg` code, `http://www.zipworld.com.au/~akpm/linux/#timepegs`